# Assignment 8: Searching for a Solution

**Overview**
The purpose of this assignment is to apply what you've learned about Problem Solving via Search. You'll use Depth first and Breadth first search to solve sudoku puzzles. While we are solving a "toy" problem, these techniques have broad real-world applications.

**More Details**
We will represent a state in the sudoku puzzle as a list of 9 lists each with 9 lists inside them. These innermost cells contain the assigned number if one has been assigned to it. Otherwise, the cell will contain a list of the remaining possible assignments. An *operation* in our approach will be to make an assignment to the **most constrained cell**, that is, the cell with the least possible remaining assignments. The **update** function will involve removing possibilities from the lists after an assignment has been made. The **goal_test** will involve checking whether all cells in the board have been assigned. We'll also include a **failure_test** method that will check to see if there are cells that have no more possibilities.

The rest of this document will guide you through completing these functions and building a sudoku puzzle solver.

**Your Job**

*The Board class*

We will represent the sudoku board in a class called *Board*.

This class has the following attributes:

- **num_nums_placed**: the number of numbers placed on the board so far (initially 0)
- **size**: the size of the board (this will always be 9, but is convenient to have an attribute for this for debugging purposes)
- **rows**: this will be a list of 9 lists, each with 9 elements (imagine a 9x9 sudoku board). Each element will itself be a list of the numbers that remain possible to assign in that square. Initially, each element will contain a list of the numbers 1 through 9 as all numbers are possible when no assignments have been made. This triply nested list will be confusing. If you want to set the value at a position in the board (make a square 7 for example) from a method you'd write something like this: self.rows[row_index][col_index] = 7. Here we're indexing into the first nested list to get the correct row, then indexing into that row to get the cell at the right column.

It should also contain the following methods (a few have already been provided, those are marked below):

- **__init__**(self) (*provided*): This should set *num_nums_placed* to 0, *size* to 9 and initialize *rows* to contain the initial representation of the board (before any numbers have been assigned). Each cell should include a list of the numbers from 1 to 9.
- **__str__**(self) (*provided*): returns a string representation of the board (yes, this will be hard to read).
- **print_pretty**(self) (*provided*): This method prints out a representation of the board with only the numbers that have been assigned. For cells that have yet to be assigned, they are printed out as * (as opposed to printing the list of possibilities).
- **subgrid_coordinates**(self, row, col) (*provided*): returns a list of (row, column) indices for all cells in the same region (the inner 3x3 grids) as the given (row, col) indices
- **find_most_constrained_cell**(self): This method should return a tuple of the row and col position (row, col) that is "most constrained". The most constrained cell should be the cell that is the list with the least number of elements (possible assignments). Note: in the case of ties return the coordinates of the first minimum size cell found.
- **failure_test**(self): returns *True* if the board has no possibility of leading to a correct solution. A good indication of whether or not it is a fail state involves seeing if any of the cells on the board contain an empty list.
- **goal_test**(self): returns *True* if the board is a complete solution, *False* otherwise. For the purposes of this solver this can be a simple function that just checks that we've placed the expected number of numbers (81 - the amount for a complete sudoku board). Note here that we're trusting our placement and update to catch invalid moves, not this method.
- **update**(self, row, column, assignment): Here *assignment* is a number to assign to the cell given by *row* & *column* arguments. It should then remove that assignment as a possibility from the candidate lists in the same row, column and 3*3 subgrid as the (row, column) index given. Remember the provided *subgrid_coordinates* method mentioned above. It should then set the given cell to the *assignment* number and update *num_nums_placed*. *Note:* we've provided a helper function called *remove_if_exists* that might be useful here. It takes a list and element to remove then removes that element from the list if it exists in the list.

Finally, outside of the board class, write a ***DFS*** function and a ***BFS*** function. These functions should utilize the *Stack* and *Queue* classes provided and follow the algorithm outlined in class. They will each take a board as input. This initial board should be added to the stack/queue. While the stack/queue is not empty, pop a board from the stack/queue to examine. If the board is a solution (*goal_test* returns *True*), return it. If a board is not a dead end (*failure_test* returns *False*) first find the most constrained cell on the board. Then try all possible assignments on that cell (adding all possible next states to the stack or queue).
*Hint:* this will involve a `for` loop over all assignments. For each assignment you'll want to create a *deepcopy* of the board before making the assignment.

Example using deepcopy:

```
>>> import copy
>>> x = [1,2,3,4,5]
>>> x_copy = copy.deepcopy(x)
>>> x
[1, 2, 3, 4, 5]
>>> x_copy
[1, 2, 3, 4, 5]
>>> x[2] = "c"
>>> x
[1, 2, 'c', 4, 5]
>>> x_copy
[1, 2, 3, 4, 5]
```

**Extensions**

Adapt your sudoku puzzle to solve the nqueens problem. That is, write a new class called chess board and think about how to formulate and solve the nqueens problem using the same approach that we used for sudoku.